NASA Technical Memorandum 4489

Open Environments To Support Systems Engineering Tool Integration: A Study Using the Portable Common Tool Environment (PCTE)

Dave E. Eckhardt, Jr., Michael J. Jipping,
Chris J. Wild, Steven J. Zeil, and Cathy C. Roberts

SEPTEMBER 1993

NASA Technical Memorandum 4489

Open Environments To Support Systems Engineering Tool Integration: A Study Using the Portable Common Tool Environment (PCTE)

Dave E. Eckhardt, Jr.
Langley Research Center
Hampton, Virginia

Michael J. Jipping
Hope College
Holland, Michigan

Chris J. Wild and Steven J. Zeil
Old Dominion University
Norfolk, Virginia

Cathy C. Roberts
Institute for Computer Applications in Science and Engineering
Langley Research Center
Hampton, Virginia

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Acknowledgments

We appreciate the help of David Green from Lockheed Engineering and Sciences Corporation, who installed and maintained the PCTE software, and Kathryn Smith, Denise Jones, Carrie Walker, and Steve Young from Langley, who provided in-house expertise with the three tools used in this study. We also want to thank John Turkovich of the Charles Stark Draper Laboratory, Inc., who provided an ASCII representation of the data flow diagrams produced by CSDL CASE.

Abstract

A study of computer engineering tool integration using the Portable Common Tool Environment (PCTE) Public Interface Standard is presented. Over a 10-week time frame, three existing software products were encapsulated to work in the Emeraude environment, an implementation of the PCTE version 1.5 standard. The software products used were a computer-aided software engineering (CASE) design tool, a software reuse tool, and a computer architecture design and analysis tool. The tool set was then demonstrated to work in a coordinated design process in the Emeraude environment. This paper describes the project and the features of PCTE used, summarizes experience with the use of Emeraude environment over the project time frame, and addresses several related areas for future research.

Introduction

Background

With the rapid development of digital processing technology, NASA programs have become increasingly dependent on the capabilities of complex computer systems. Current flight control research, which advocates active controls (ref. 1) and fully integrated guidance and control systems (ref. 2), relies heavily on digital processing technology. These advanced guidance and control systems, designed to optimize aircraft performance, will demand high-throughput, fault-tolerant computing systems. Additionally, safety concerns will dictate that future generations of commercial aircraft have hardware and software systems with extremely low failure rates such that catastrophic failures are extremely improbable, that is, such failures are "not expected to occur within the total life span of the whole fleet of the model (ref. 3)." The functional performance, reliability, and safety of these systems are of great importance to NASA; thus, a component of the research within the NASA Aeronautics Controls and Guidance Program is directed toward the development of design, assessment, and validation methodologies for flight-crucial systems (ref. 4). An important aspect of this work is developing the engineering tools that support cost-effective certification of future flight systems.

The state of the art of this technology was a primary issue of discussion at a workshop on digital systems technology held at Langley Research Center. The consensus of this representative sample of the U.S. aerospace industry was that there is a "lack of effective design and validation methods with support tools to enable engineering of highly integrated, flight-critical digital systems (ref. 5)." Design methods are generally fragmented and do not support integrated performance, reliability, and safety analysis. There is a growing recognition that such integrated studies will require an integrated design and evaluation environment. A primary purpose of such an environment is to achieve a level of *integration* of the diverse support tools used in the system development. Ideally, the environment is "open," as distinguished from "proprietary," in which case the integration of foreign tools is difficult at best.

The integration function of a computer-aided software engineering (CASE) environment can be split into three areas: data, control, and presentation integration (ref. 6). In addition to these areas, a fourth area, which deals with process integration, is emerging as a critical functionality that can also be provided by the environment (ref. 7). Data integration can be achieved by exchanging data between tools directly or by storing the data in a shared project directory. A central repository for project information facilitates configuration management and tends to define project information structures that are independent of the specific tools used to manipulate this information. Control integration allows tools to coordinate their activities to maintain consistency between the information managed by each tool. A well-known example is the UNIX **makefile** system, which ensures that an executable program is generated from the latest versions of the source code and the "include" files. The purpose of presentation integration is to provide a uniform user interface to the services provided in the environment. Process integration deals with supporting the dynamics of software development by defining, managing, and certifying the set of activities across the software life cycle.

Generally, the two approaches used to achieve tool integration are tool collections and an integrated project support environment (IPSE). The tool collection approach represents the state of practice.

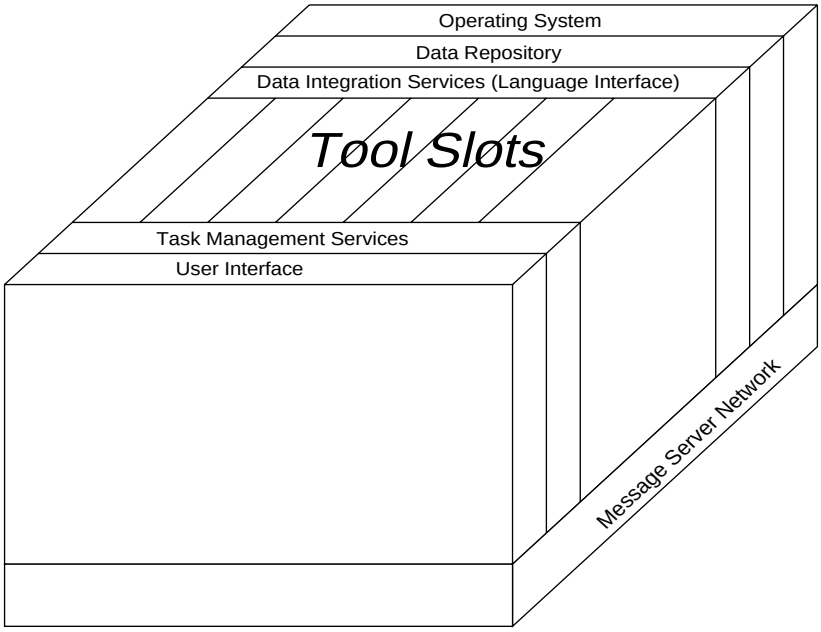

Figure 1. IPSE reference model.

Here, the emphasis is on the tools themselves. This approach acknowledges a variety of tools on the market with a variety of mechanisms for working together and identifies the lowest common denominator of services for tools and support-specific end-user activities. The services provided and activities supported are typically no more than what is provided by the operating system.

An IPSE provides a common infrastructure into which tools can be embedded. The reference model for an IPSE is the "toaster" model shown in figure 1 (ref. 7). This model defines a set of services within a framework. The message-server network allows communication between different tools and services in the environment. Typically, this service builds on or extends the communications services provided by the underlying operating system. The user interface is typically provided by one of the emerging window management standards such as Motif (ref. 8). Task management, data repository, and data integration services are provided by the IPSE. By fitting into the "slots" of the toaster model, the tools are integrated and can work together efficiently.

The largest roadblock to integration is a lack of widely accepted standards. Vendors have invested time and money into their own integration techniques and move slowly to discard or revamp their investment for standards that are not yet widely accepted. The result is a host of integration "standards," very few of which are compatible with each other. An overview of major standardization efforts can be found in reference 9. Of the tool-oriented standards, a recent standard from the Object Management Group (OMG) has emerged with implementations by Sun Microsystems, Inc. and Hewlett-Packard Co. (ref. 10). Of the IPSE standards, the Common Ada Programming Support Environment (APSE) Interface Set (CAIS) (ref. 11) and the Portable Common Tool Environment (PCTE) (ref. 12) are two standards that address the whole IPSE reference model. The PCTE is a European Computer Manufacturers Association (ECMA) standard tool-building framework that is gaining widespread support both in Europe and the United States.

Objectives

A primary research thrust of the Systems Architecture Branch at Langley Research Center is to develop the computer-aided technology for safety-critical software and high-performance architecture systems for advanced aircraft avionics. This work is motivated by the belief that computer automation techniques are eminently possible through focused research on application-specific domains and that these automation methods will result in significant gains in productivity, quality, and safety. To support this research, a project was initiated to evaluate an open environment software infrastructure as the framework for this design technology. This paper describes the use of the PCTE version 1.5 Public Interface Standard as implemented by Emeraude, a

French company. This project emphasized the Object Management Services (OMS), by far the most significant feature of PCTE, and the tool encapsulation facilities of the Emeraude environment.

The long-range goal of this work is to define the role and requirements of an open framework for developing highly integrated, flight-critical computer systems. Frameworks are vaguely defined but generally refer to environments for the communication and integration of tools in a process (ref. 13). Accommodating the entire design process is a recent emphasis of these environments, as opposed to the previous emphasis on the tools themselves, which were often tools with proprietary interfaces. To better understand the role that an open environment can play in a tool integration context, a study was conducted in which three existing software products were encapsulated and used in a coordinated design process. These tools had not previously been used in a coordinated manner. Two of these products, CSDL CASE (ref. 14), a computer-aided software engineering design tool from the Charles Stark Draper Laboratory, Inc., and InquisiX (ref. 15), a software reuse tool from Software Productivity Solutions, Inc., are alpha-release versions. The third tool, ADAS (ref. 16), an architecture design and analysis tool, is a commercial-off-the-shelf tool. The objectives of the study were to

1. Demonstrate through a simple but realistic example the value of an open environment in facilitating a coordinated design process; because members of the project team did not have previous experience with the Emeraude environment, a small demonstration project was the fastest way to confront open environment issues in the tool integration context

2. Demonstrate the encapsulation and integration of existing software tools through a shared, common infrastructure; although productivity and reliability advantages exist for building new tools in an open environment framework, many existing tools serve useful functions and are not likely to be replaced in the foreseeable future

3. Identify areas for further research in the open environments that are needed to support the development of automated design technology

Project Description

Demonstration Context

To meet the objectives outlined for this study, part of the study reported in reference 17 was reproduced and automated. That study examined the performance of various architectures for large-grain data flow parallelism. The architectures studied involved an array of processors interconnected to a scheduler and a work load generator. A typical architecture is shown in figure 2. The performance of an architecture was tested using a data flow graph depicting the major software processing elements and the order of execution as determined by the data flow between processing elements. Previously, the data flow graph was converted by hand into a textual representation that was read by the **SpawnProcess** component shown in figure 2. The hand generation of different data flow work loads was one of the most burdensome tasks in the original study. Although CASE tools were available to generate these data flow diagrams, the outputs of the tools were not compatible with the inputs to the simulation tool used in the performance analysis. An objective of the project, then, was to investigate the direct use of the output of an existing CASE tool used by software designers as input to the simulation tool used by the architecture designers. Essentially, the architecture studies would have the benefit of using real software work loads, although for this study, the actual work loads defined in reference 17 were used.

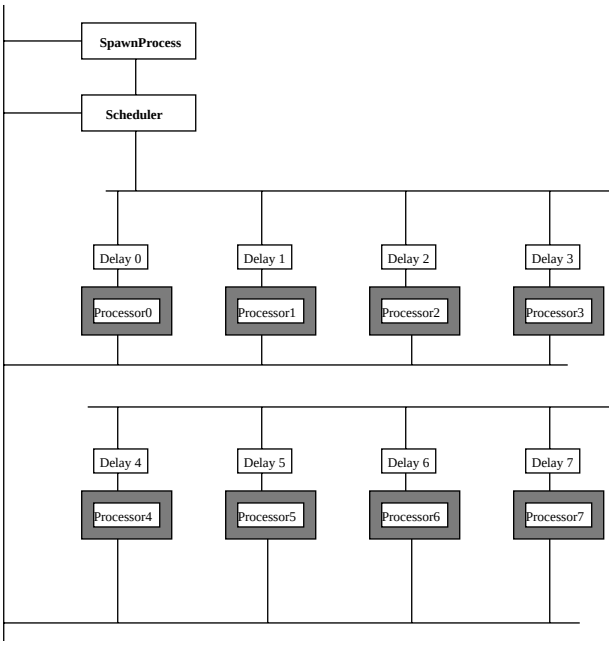

Figure 2. Macro data flow scheduler (8 processors).

An overview of the demonstration project is shown in figure 3. For this demonstration, the **Software Designer** generates the software work load data flow graphs using the CSDL CASE software design tool. These diagrams are used to generate different textual representations of work loads for input to the ADAS simulation tool. Because many work loads could be generated for testing the

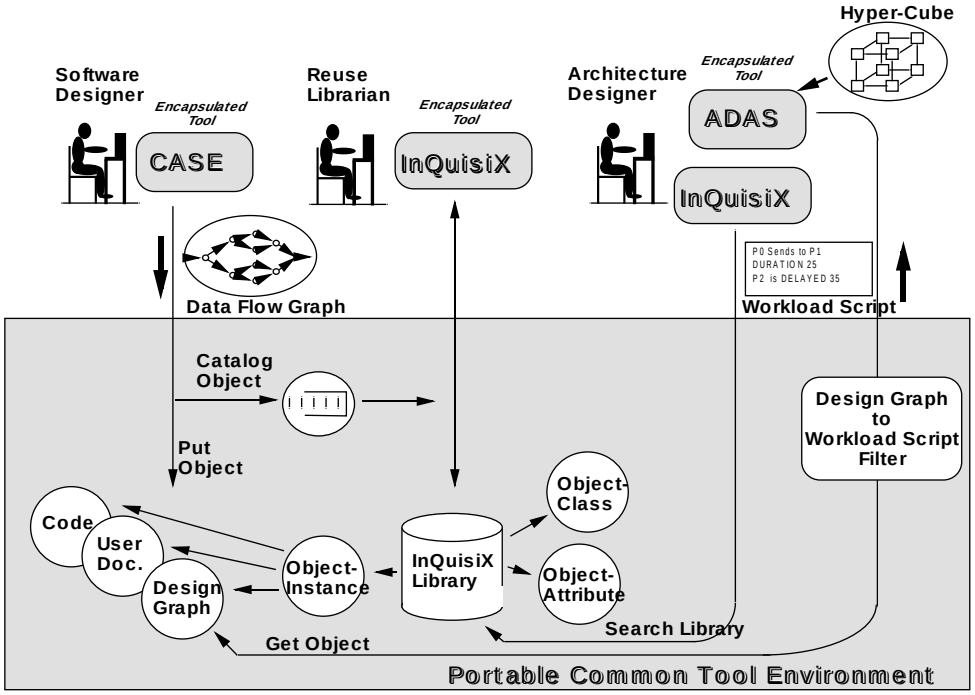

Figure 3. PCTE demonstration.

performance of proposed architectures, ideally the work loads would be classified and stored in a reuse library. The application of the InQuisiX-reuse library tool would further demonstrate the capabilities of the open environment for integrating engineering tools. After the work load information was cataloged by the **Reuse Librarian**, the InQuisiX tool could be used by the **Architecture Designer** to browse the data base and select the work load with the desired attributes to use with the ADAS tool.

Tool Set

The CASE tool used for generating the data flow diagrams was developed by the Charles Stark Draper Laboratory, Inc. (CSDL) under contract to Langley Research Center. This tool is oriented toward the aerospace controls engineer and can generate Ada code and documentation directly from engineering block diagrams of the control algorithms. Figure 4 shows a block diagram for a yaw-damper algorithm consisting of first-order lags (FOLAG), washout filters (WOUT), switches (SWITCH), and limiters (lim), which can all be retrieved from a library. As originally developed, the CSDL CASE output consists of the automatically generated Ada code and supporting documentation. The engineering diagrams are stored in internal libraries and are not available for other engineering tools. One of the first tasks was to specify a generic data flow representation and to task CSDL to produce this format from the internal representation within the CSDL CASE tool. When this task was accomplished, the data flow diagrams could be used with other engineering tools.

The availability of InQuisiX, developed by Software Productivity Solutions, Inc. under Small Business Innovative Research (SBIR) contracts, offered another dimension for study within the scope of this project. InQuisiX can define a taxonomy for a set of objects and store and retrieve objects according to this classification scheme. Many of the features of the classification scheme of InQuisiX are available as part of the object management facilities of PCTE. However, InQuisiX does provide a user interface to search the object base for those objects that match user-specified criteria. A comparable searching facility was not available in the Emeraude implementation of PCTE, so InQuisiX was selected for the study to provide this capability. In discussions of the role of InQuisiX for this project, it was generally felt that the development of InQuisiX would have been greatly simplified if access to the common services provided by PCTE were available.

ADAS is a discrete-event simulation tool marketed by CADRE Technologies, Inc. This tool allows the user to graphically define a system model, run a discrete-event simulation of the system, and view the simulation as it progresses. Additionally, ADAS also provides tables of the results that can be further analyzed. This tool has proven to be effective

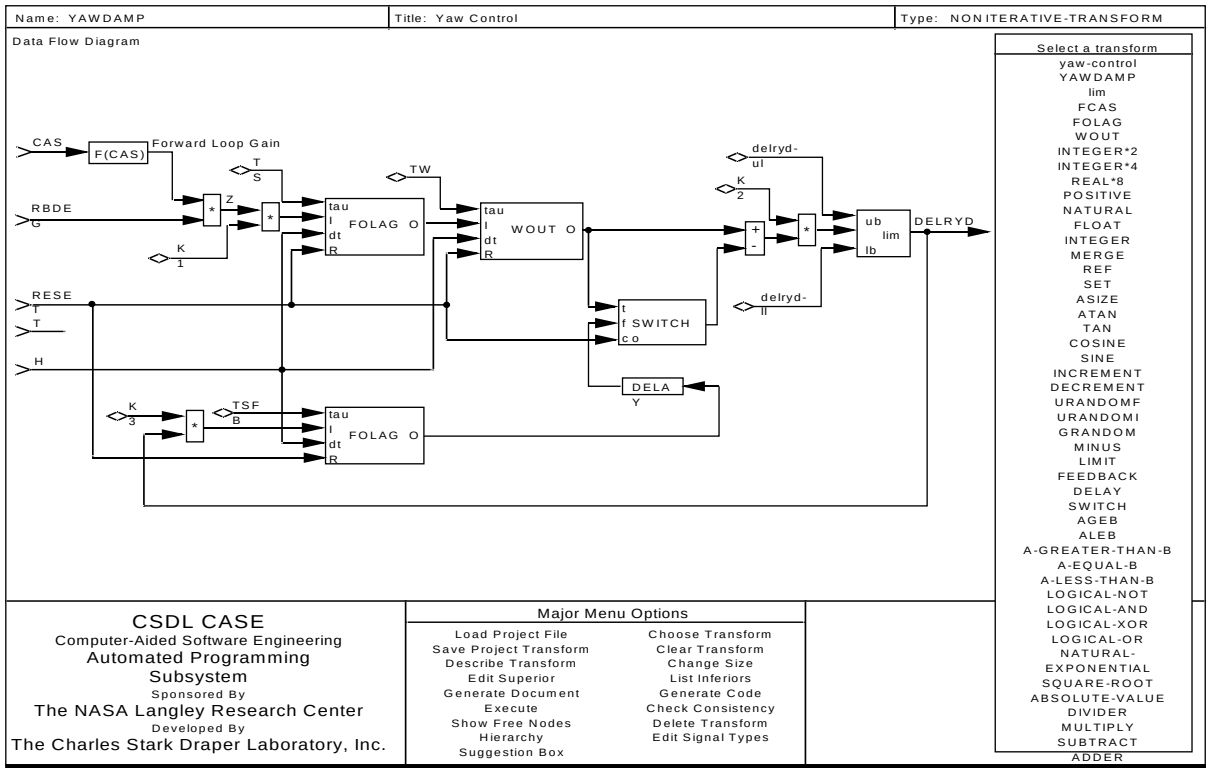

Figure 4. CSDL CASE engineering block diagram.

for measuring the performance of proposed parallel architectures for aerospace applications (ref. 17).

Project Implementation Using PCTE

PCTE Services

Figure 5 illustrates the following major services offered by the Emeraude implementation of PCTE version 1.5:

1. The most significant aspect of PCTE is the *Object Base*, which is the common repository of all data in PCTE. The Object Base is a typed, persistent store.

2. The *Metabase* is that portion of the Object Base devoted to describing the contents of the remainder of the Object Base. In practice, the Metabase is a collection of objects that describes the data types of the objects in the Object Base.

3. The primary operations for accessing the Object Base are provided by the PCTE *Object Management System* (OMS). The OMS provides tools that can create, examine, and alter objects in the Object Base.

4. The *Execution/Communication* services include support for distribution of the Object Base and for interprocess communication.

5. The *Metabase Services* are operations that use the OMS to examine and update the Metabase. Examples include operations to create new types and to determine the type of an object.

6. *Version Management Services* are available for all objects in the base.

7. *Data Query Management Services* allow programs to formulate searches of the Object Base.

Not all the services listed above were used in this project. The Execution/Communication services were largely irrelevant because the evaluation copy of the Emeraude environment obtained for this project was limited to a single network node. The Data Query Management facilities currently lack an interactive interface in the Emeraude environment, thus appear to be relatively inaccessible. Version Management, although critical to long-term projects, would not have been fully exercised during this relatively short project. On the other hand, the most novel and pervasive new capability offered by typical open environments is an Object Base. The PCTE Object Base and OMS were used extensively. For this application, Metabase services were employed toextend the Metabase, which added new data type descriptions in accordance with data manipulation by the project tool set.

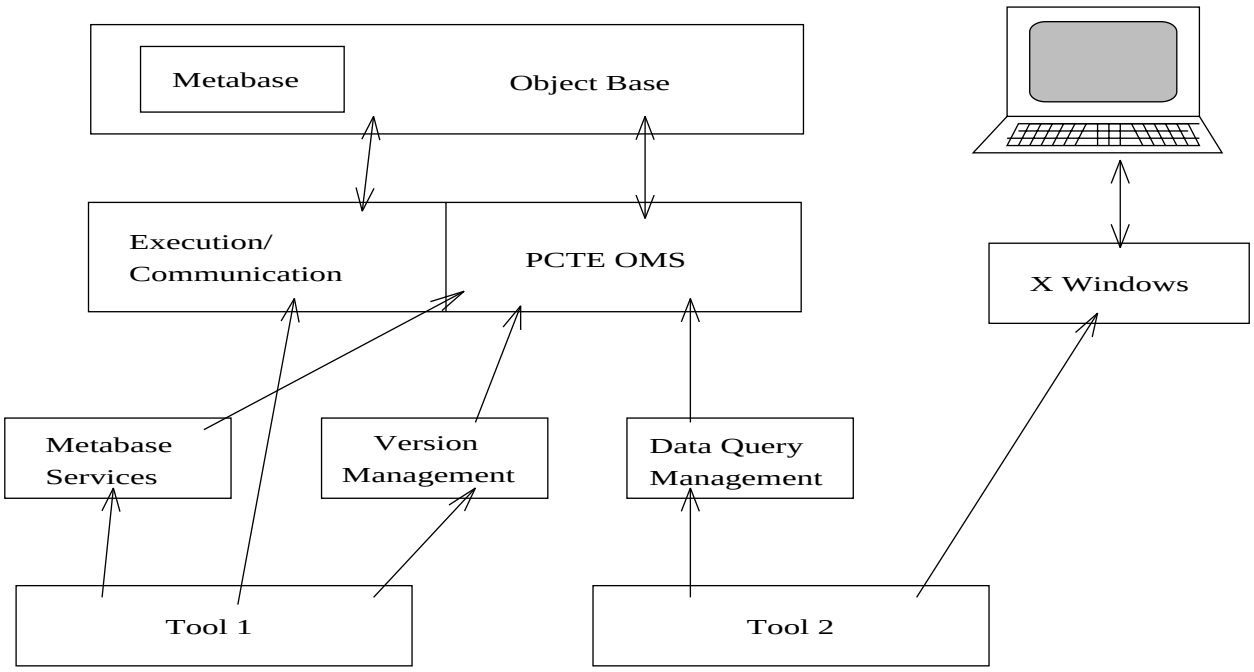

Figure 5. PCTE services.

Tool Classes

From the viewpoint of PCTE, two important classes of tools are presented, as illustrated in figure 6.

Native: Native tools are those designed and implemented with specific PCTE support. Such tools will ideally distribute inputs and outputs across many OMS objects, attributes, and relations in an effort to anticipate the information requirements of other tools that may later be added to the environment. To native tools, the OMS represents an elaborate storage system supporting interobject relations.

Foreign: Foreign tools are those designed for use in another environment such as UNIX. Such tools expect inputs and outputs to appear in simple files.

Foreign tools (such as UNIX tools) can be imported into the Emeraude environment by a process of *encapsulation*. The encapsulated tool still receives inputs and outputs from "files," but many of these files are now objects of type `file` (or some subtype of `file`) in the Object Base. The OMS allows the environment to record information about these files as attributes and relations without examining the internal file structure. The file objects themselves are treated as black boxes by the Emeraude environment.

Emeraude provides two mechanisms for encapsulation. The first is to recompile the tool, substituting the Emeraude I/O library for the "conventional" UNIX I/O library. This substitution provides a set of I/O operations with signatures that are identical to the file-handling primitives of UNIX, but that actually open, close, read, or write objects of type `file` in the PCTE Object Base. The second mechanism, which is useful for tools that receive their file names via their command line invocation, is to wrap a simple Emeraude shell script around the tool invocation. Within that script, the command line parameters are processed by a special Emeraude command to convert the logical paths to file objects into the actual UNIX path names where the Object Base has located the particular file objects. Path names can then be read and/or written using the normal UNIX primitives.

The tool set used for this study consisted of foreign tools. Because the source code was not available for these tools, the second encapsulation method was employed in this project. However, the CSDL CASE tool, as previously mentioned, was modified by the CSDL to use internal information. This information represented the data flow object. Although the encapsulation method was used with this tool, it had some of the characteristics of a native tool.

The OMS Type System

As noted earlier, all objects in the PCTE Object Base are typed. The type determines

1. Attributes that describe the object

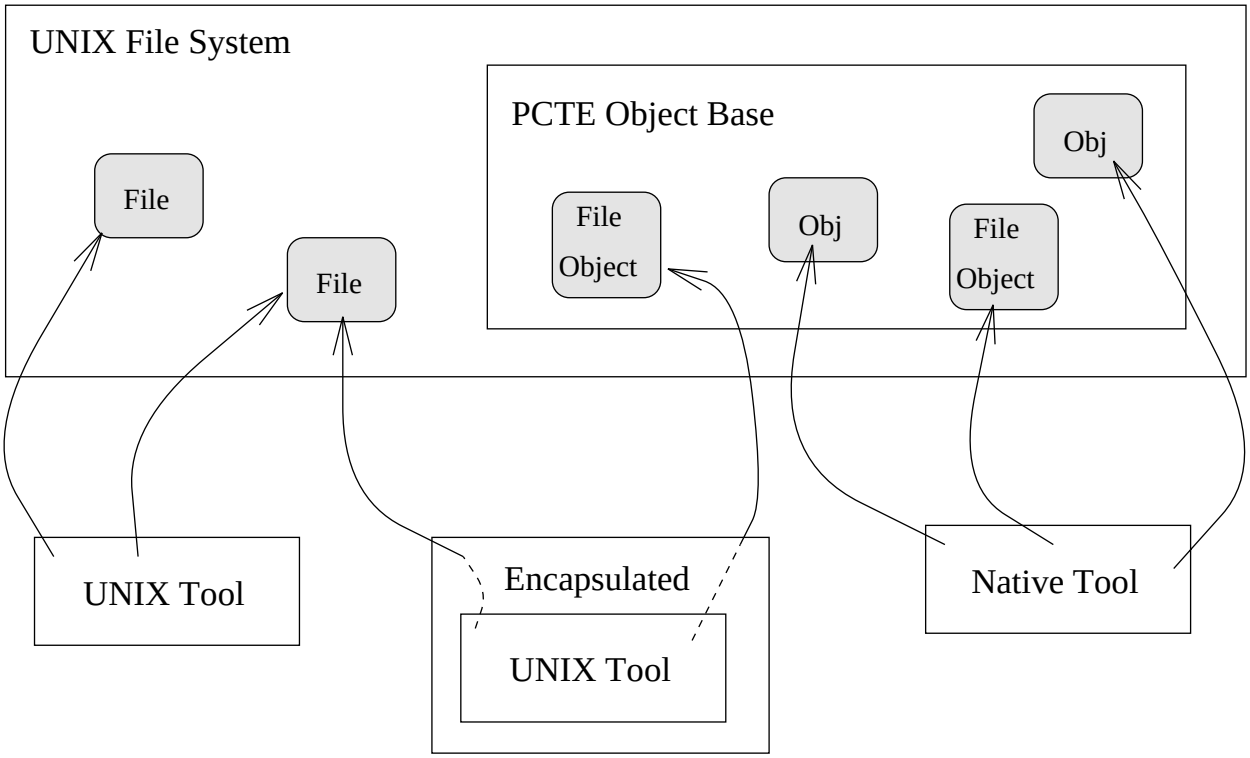

Figure 6. Integrating tools.

2. Relations (links) the object may have with other objects

3. Whether the object has *contents* (i.e., can we open and/or close it and apply read and/or write operations to it?)

An *attribute* is a named value associated with an object. Attributes can be strings or numbers. A *relation* is a bidirectional link between two objects. Each direction has a different name. Both attributes and relations can be viewed as "properties" of the object. When that property is itself another object, it is a relation. When the property is a simple string and/or integer value lacking a separately addressable identity, it is an attribute.

Types are related by inheritance, which means that if `Sub` is a subtype of `Super`, then all attributes and/or relations of `Super` are also available for objects of type `Sub`.

The Emeraude environment comes with a number of predefined types. These types define OMS analogs of the following familiar concepts:

`dir`	a "container" of files and other directories; more precisely, an object that serves as the head of a number of links to directories and files
`file`	an object that has "contents" and can be written to and/or read from; has attributes: owner and modification date, among others
`object_code`	a subtype of file, intended to hold only object code
`c_source`	a subtype of file; to the usual file `attributes` and `relations` adds `links` to "include" files and other C language-specific information
`object`	the "root" of all types

Figure 7 shows the inheritance relations that relate the predefined types. The inheritance hierarchy is important to determine the properties offered by objects of any given type. For example, any `object` has a `name` attribute; therefore, a `file` has a `name` as well. A `file` has contents; therefore, so does any `c_source` object. On the other hand, `c_source` objects have attributes and relations that are specific to `c_source` code and would not be applicable to general `files`.

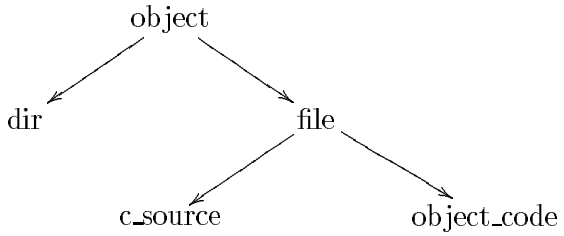

Figure 7. Predefined types: inheritance hierarchy.

Because each object is typed, the environment and the tools running in that environment are aware of what attributes and relations are available for any given object. The environment can prevent the use of inappropriate attributes and relations with an object. Less obviously, the type system allows control of the visibility of objects, attributes, and relations. Each user has a *working schema*. The working schema is a list of object, attribute, and relationship types available to the user. Attempts to access an object, attribute, or relation whose type is not in the working schema will fail, just as if that object, attribute, or relation did not exist. Individual users and groups can be given or denied access to sets of types, thus given or denied access to objects of those types.

Type Schemas

Types are grouped into *Schema Definition Sets* (SDS). A type may appear in the SDS's. As new tools are brought into the environment, new kinds of input/output data employed by those tools must be described to the environment. An environmental description is accomplished by defining a new schema containing the data types needed by the new tool.

As an example of this design process, consider the problem faced in this project of integrating the CSDL CASE and ADAS tools. This scenario called for software designs (data flow diagrams) from CSDL CASE to be combined with machine-characteristic information to produce a work load script to drive an ADAS simulation.

This problem suggests an initial list of new types: a data flow diagram, machine characteristics, and a work load script.

On closer examination, it was determined that CSDL CASE can represent data flow diagrams as directed graphs or as a text script, suggesting two more types: `dfd_graph` and `dfd_script`.

The first step in defining these types was to organize them into an inheritance hierarchy, as shown in figure 8.

Objects of type `dfd_script` and `dfd_graph` are produced as files by CSDL CASE; they have contents (i.e., we must be able to read and write them), so it makes sense that they should be treated as subtypes of `file`. Similar arguments hold for `machine_char` and `workload_script`.

The notion of a data flow diagram (`dfd`) as a possible combination of graph and script is an organizational idea (for example, analogous to a directory). As such, this flow diagram has no contents of its own so it cannot be a file.

After the inheritance hierarchy has been set, relations are added among the types. Relations serve both to add information about the objects and especially to enforce certain constraints:

1. For every work load script, there can be only one data flow diagram and one machine-characteristics file.

2. The same data flow diagram can be used to produce many different work loads (e.g., by varying the machine characteristics and/or number of concurrent tasks).

3. The same machine characteristics can be used to produce many different work loads (e.g., by varying the software data flow and/or number of concurrent tasks).

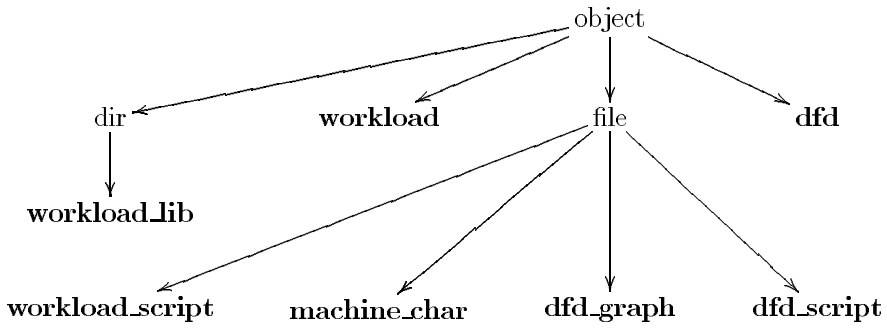

Figure 8. Project inheritance hierarchy.

4. For any data flow diagram, there can be at most one CSDL CASE graph and at most one CSDL CASE script.

The first three constraints can be seen in the schema shown in figure 9. In this figure, the boxes denote types, and the triangles and diamonds denote relations that may link objects of the indicated type. A diamond is used when a name is assigned to each direction of the relation, and a triangle is used when a name is given to only one of the two directions. Thus, for example, from any work load object, one can follow a `.script` link to find the corresponding work load script, and from a work load script object one can follow a `.script_for` link back to its work load.

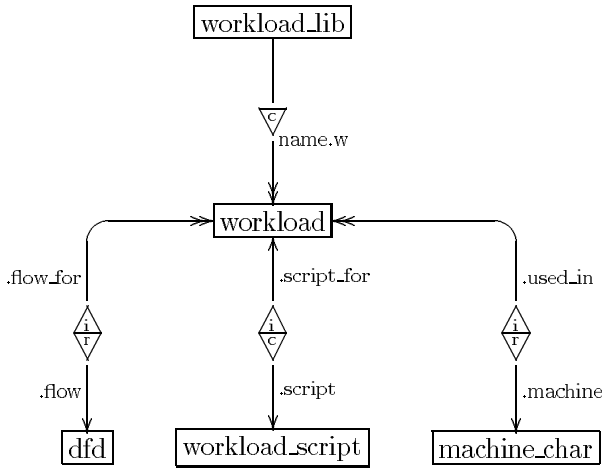

Figure 9. Work load schema definition set.

Links that end in a single arrowhead denote a many-to-one link. Links ending in a double arrowhead denote a many-to-many link. Thus it is apparent from figure 9 that a given work load has a single data flow diagram (via the `.flow` link); however, each data flow diagram can contribute to many work loads (by way of the `.flow_for` link). When both directions of a relation are many to one, the combination is equivalent to a one-to-one relationship. Thus, the relation between work loads and work load scripts is one to one.

The fourth restriction is captured in the schema shown in figure 10, which also illustrates an early decision that the environment might contain many different tools capable of building and manipulating data flow diagrams such as both the CSDL CASE and ADAS tools.

The final step in developing schemas for describing tool interactions is to "decorate" the object types with attributes to help describe the objects and to make internal information available to other tools. Some of the attributes we employed for `workload` and `dfd` objects were

`name`—an identifier inherited from the root type `object`

`topology`—a name describing the general shape of a data flow graph

`num_tasks`—the number of duplicate tasks, each a complete instance of the software data flow diagram

`width, max_path_length,`...—various attributes describing the shape and properties of the `dfd` graph

Note that information such as the `dfd` and machine characteristics used with each work load is already available, but as relations, not attributes.

By following relationship links and examining the attributes of the objects encountered, a variety of searches and retrievals can be performed. Emeraude has query and searching primitives (the Data Query Management Services), but these primitives are provided as a library of C routines. No interactive tool except a basic OMS browser is currently provided. InQuisiX, which was used for this purpose, is a reuse librarian tool that describes library units in terms of attributes and permits interactive searches for units that satisfy various constraints on those attributes. Many attributes defined for work loads were chosen to illustrate the processes of registering a work load in a reuse library and of permitting later searches and retrievals of those work loads. In such a situation, we would anticipate that some, but not all, of the useful information about the work load would be assigned by the tools that created the work load. This assignment by the tools is true of (1) the name, (2) the links to data flow diagram, work load script, and machine-characteristics files, and (3) the number of concurrent tasks. Other attributes, primarily those concerned with documentation, would be filled in by the reuse librarian when the object is cataloged for general use.

Thus, in this case, it was necessary for PCTE attribute values to be sent to the InQuisiX library, and for any changes to object attributes made by the InQuisiX librarian to be reflected later as PCTE attributes.

Summary of Experience With Emeraude PCTE Version 1.5

The project was undertaken over a 10-week period from June 1, 1992 to August 7, 1992. This period was chosen to correspond with the summer on-site performance period of the JOint VEntures (JOVE) program funded by Marshall Space Flight

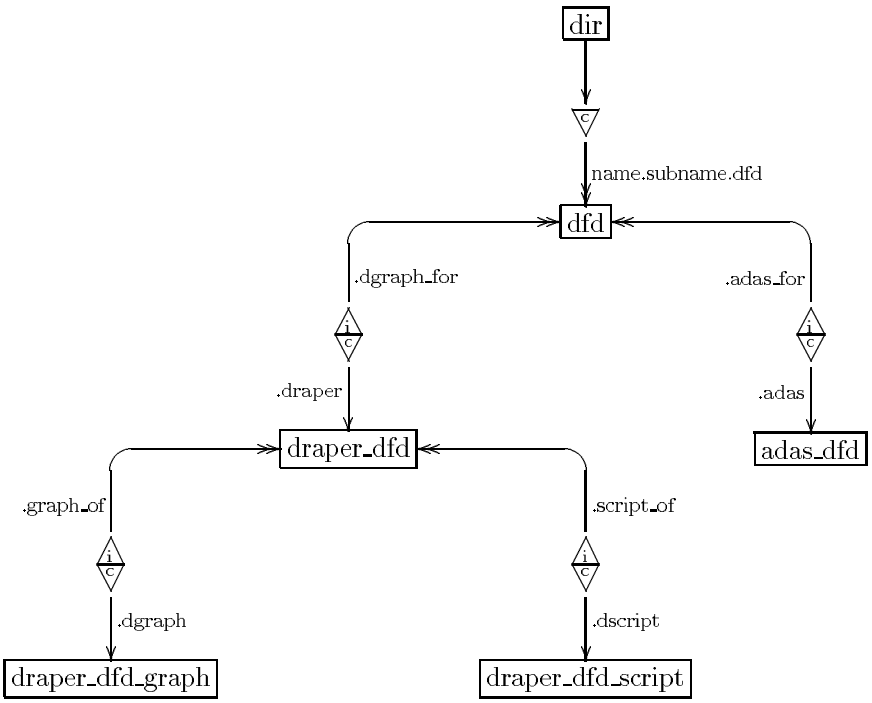

Figure 10. Set for dfd schema definition.

Center, which sponsored one project member. The project, which was successfully completed within the allotted time, consisted of four major tasks:

1. Education (2 weeks): This task included installing and studying the capabilities of the Emeraude environment. Additionally, the target tool set was unfamiliar to the project team; therefore, part of this time was devoted to studying the capabilities of the two alpha-release tools and the commercial tool.

2. Demonstration definition (3 weeks): During this time the functions to be demonstrated (software reuse, CASE, and architecture performance evaluation) were refined, and the demonstration process was defined. These refinements involved evaluating the data integration possibilities of the target tool set. The need for a CASE tool modification was recognized, and the specific modification was defined. This modification was implemented in a very short time by the developers of the CASE tool at the Charles Stark Draper Laboratory, Inc.

3. Implementation (3 weeks): The PCTE object types, relations, and schemas were developed with the tool encapsulation scripts and filters.

4. Evaluation (2 weeks): During the last 2 weeks, the coordinated design process, as represented in figure 3, was demonstrated. The project was also documented during this time.

The fact that this project was completed within the 10-week period indicates the relative ease with which foreign tools could be encapsulated in the Emeraude implementation of PCTE. The facility of this effort was partly due to the orientation of the environment toward UNIX and the project team's knowledge of UNIX.

The current PCTE standard only supports large-grain data modeling at the object level. The internal structure of objects is treated as a black box by PCTE; thus, tools designed for manipulation of the contents of the objects must agree on format outside the modeling capabilities of PCTE. Nevertheless, the ability to model objects at the large-grain level and to define the relationships between them was valuable. The development of an explicit object data model clarified the role of each tool in the project's engineering development process and the relationships between the tools. This model also enforced consistency and precision in the use of the information defined and manipulated by the tools. Although PCTE does not support the encapsulation of object types with behavior, it was not limiting for this particular project.

Additionally, the availability of an external ASCII format for the internal contents of the data flow diagrams and the work load scripts allowed some fine-grain data manipulation through the development of simple filter programs that translated between formats. Most of this project was developed using the facilities of the Emeraude Shell Programming

Language and Makefile facility. Programs in C were written only for the fine-grain data filtering.

Because the three tools used in this project were developed by three different vendors with different window systems and approaches to user-interface management, each tool presented a distinct interface to its services. The availability of an open environment with infrastructure support for defining a user interface would contribute greatly toward providing a consistent "look and feel" for each tool.

One major advantage of the PCTE Object Management System is support for transaction control. The user could define the beginning of an activity, manipulate a set of objects, then abort the activity and roll the system back to the original state. This facility eased the tasks of learning PCTE and correcting the software developed for the project because the Object Base could always be returned to a consistent state. In fact, part of the demonstration was performed as a transaction activity, then rolled back to return the system to the same starting state for subsequent demonstrations.

Research Issues

The study also considered several areas of future research, which include object management support with programming language interfaces, development environments for concurrent systems, and process modeling.

Object Base Technology

The object-oriented data base (OODB) is a relatively new class of data storage that has not yet matured to the same degree as more conventional data base forms. Open issues affecting both the features and performance of OODB's include: type evolution, scale and granularity, inheritance of behavior, and efficiency. Many of these issues are discussed in references 18 and 19. The development of appropriate type systems for persistent object management in particular is complicated by the need to map persistent object types into types that can be processed by a variety of conventional programming languages; that is, the purpose is to achieve an *interoperable* type system. Most prior efforts to achieve interoperability have been directed at overcoming differences in machine and/or language representation of "equivalent" data structures. Approaches have included

1. Imposing a single data model: An example is the widespread adoption of the IEEE standard for floating-point number representation. By encouraging all vendors to comply with this model, interoperability of this data form is achieved. Unfortunately, by its nature this approach can only be achieved for a finite number of data structures.

2. Imposing a unifying data model: Data description languages such as the Interface Description Language (IDL) provide a uniform model for construction of new compound data structures from a small set of primitives (ref. 20).

The combination of single data models for primitives (for example, numbers and characters) and a unified model for construction of compound structures is an important step toward achieving interoperability. There is another level, however, at which it is often more convenient to consider the issue: the level of the data abstraction implemented by a particular representation. Representation-level schemes provide only minimal assurance that a given data structure will be manipulated in an acceptable manner by users from distinct environments. In other words, the data are transferred, but the enforcement of the abstraction captured by that data is left to the good will and capabilities of the programmers in each environment.

Programming Language Interfaces to Persistent Data

Conventional data base languages have long been criticized for a lack of programming power and expressiveness as well as being far behind the state of the art in incorporating software engineering concepts into language design. On the other hand, traditional programming languages offer no support for persistence beyond the idea of a "file." For this reason, interest has been growing in "persistent programming" languages that merge the expressiveness of modern programming languages with support for persistent object stores (ref. 21).

Most persistent programming languages organize persistent data into special "bulk" data types such as sets or relations (refs. 22 and 23). A smaller number of these languages have attempted to merge persistence support directly into a traditional language with little or no visible change to the language (ref. 24). Curiously, this minimally intrusive approach has seldom been employed with languages that offer rich support for data abstraction. An exception is Zeil's `ALEPH` project which adds persistence to Ada (ref. 25 and work done for Langley under NASA grant NAG1-439 and National Science Foundation grant CCR-8902918 at Old Dominion University Department of Computer Science). The `ALEPH` language preprocessor currently under development can serve as a vehicle for experimentation

and distribution of these protocols by providing a simple interface to persistence and garbage collection for Ada programmers.

Environments for Concurrent Systems

Concurrent software design differs from sequential software design in several significant respects. A major difference between techniques involves the coordination between processes. In a concurrent system, processes must communicate, and the semantics of such interprocess communication can easily be utilized incorrectly by the implementation. The results are concurrency anomalies such as deadlock or corruption of shared data. The construction of concurrent system design tools requires a model of concurrent systems that lends itself to concurrent system design and exploitation of that model in a specification language that captures the model properties. Early work on models of concurrent systems emphasized avoidance of system execution anomalies (for example, deadlock) and guaranteed the correctness of shared-data access. Recent work focuses more on efficient forms of concurrency that can be derived by redefining correctness properties and allowing for varied and more complex interleaving of shared data operations. Early work focused on enforcing correctness criteria at the process level rather than the individual operation level; recent work emphasizes individual operations.

One model, the general process model (GPM), provides a framework for the consideration of both system syntax (that is, the arbitrary forms of data objects and accesses) and semantics (that is, the meaning and effect of individual data objects and manipulations). This model is the basis for the development of environments for designing and analyzing concurrent systems. (For example, see ref. 26.)

Process Modeling and Management

The validation and verification of mission-critical computer systems must encompass not only the artifacts produced (for example, specifications, code, and designs) but also the process used to develop those artifacts. Process modeling refers to the definition of the set of activities that comprises the development process and the interrelationship of these activities. Process management refers to the use of a process model in controlling, measuring, and certifying the dynamics of development. The importance of process modeling for the development of complex computer systems has been recognized for some time (ref. 27), but interest in this area has increased rapidly over the past few years (refs. 7 and 28).

An open environment can play a significant role in the management, enforcement, and documentation of the development process. Because a process model controls the set of activities that makes up the engineering development, this process is best embedded in a unified development environment in which all access to development resources can be monitored, recorded, and controlled.

Conclusions

The development and operation of complex computer systems will require computer-aided support throughout the system life cycle. The proliferation of computer-aided software engineering systems in the past decade is a testimony to the widespread need for automation technologies to support computer system development. Although the specific set of technologies, tools, and methodologies varies with application and state of practice, it is possible to identify an underlying infrastructure that provides the basic set of services necessary to support a unified system development environment. An environment such as the Portable Common Tool Environment (PCTE) provides this underlying set of services.

This study investigated the role that an open environment could play in the development of mission-critical computer systems. A conceptual design scenario for the performance evaluation of parallel computer architectures that involved three diverse software tools was proposed. The tools were integrated using the emerging PCTE standard, and the design scenario was successfully demonstrated.

The study demonstrated the feasibility of integrating a set of independently developed tools into a design environment and suggests that the current state of open environment standards, as represented by PCTE version 1.5, is sufficiently mature to warrant consideration in future implementations. This study also suggested *future* tool development should be undertaken within an open environment context and that *existing* tools should be migrated into that environment. This strategy would provide many opportunities for the integration of existing and proposed capabilities into a unified and manageable development process.

NASA Langley Research Center
Hampton, VA 23681-0001
June 30, 1993

References

1. Grantham, William D.; Person, Lee H., Jr.; Brown, Philip W.; Becker, Lawrence E.; Hunt, George E.; Rising, J. J.; Davis, W. J.; Willey, C. S.; Weaver, W. A.; and Cokeley, R.: *Handling Qualities of a Wide-Body Transport Aircraft Utilizing Pitch Active Control Systems (PACS) for Relaxed Static Stability Application.* NASA TP-2482, 1985.

2. Cronin, M. J.; Hays, A. P.; Green, F. B.; Radovcich, N. A.; Helsley, C. W.; and Rutchik, W. L.: *Integrated Digital/Electric Aircraft Concepts Study.* NASA CR-3841, 1985.

3. Waterman, Hugh E.: FAA's Certification Position on Advanced Avionics. *Astronaut. & Aeronaut.*, vol. 16, no. 5, May 1978, pp. 49–51.

4. Holcomb, Lee; Hood, Ray; Montemerlo, Melvin; Jenkins, James; Smith, Paul; DiBattista, John; De Paula, Ramon; Hunter, Paul; and Lavery, David: *NASA Information Sciences and Human Factors Program—Annual Report, 1990.* NASA TM-4291, 1991.

5. Meissner, C. W., Jr.; Dunham, J. R.; and Crim, G., eds.: *NASA-LaRC Flight-Critical Digital Systems Technology Workshop.* NASA CP-10028, 1989.

6. Wasserman, Anthony I.: Tool Integration in Software Engineering Environment. *Software Engineering Environments*, F. Long, ed., Volume 467 of *Lecture Notes in Computer Science*, Springer-Verlag, 1989, pp. 137–149.

7. *Reference Model for Frameworks of Software Engineering Environments.* NIST SP-500-201 (ECMA TR/55, 2nd ed.), National Inst. of Standards and Technology, Dec. 1991. (Available from NTIS as PB92 158 328.)

8. *Open Software Foundation: OSF/Motif Programmer's Guide.* Prentice-Hall, c.1990.

9. Zarrella, Paul F.: *CASE Tool Integration and Standardization.* CMU/SEI-90-TR-14 (Contract F19628-90-C-0003), Carnegie-Mellon Univ., Dec. 1990. (Available from DTIC as AD 235 640.)

10. *The Common Object Request Broker: Architecture and Specification.* OMG Doc. No. 91.8.1, Digital Equipment Corp., Hewlett-Packard Co., HyperDesk Corp., NCR Corp., Object Design, Inc., SunSoft, Inc., c.1991.

11. *Military Standard—Common ADA Programming Support Environment (APSE) Interface Set (CAIS), (Revision A)*, Volumes I–IV. MIL-STD-1838A, Apr. 6, 1989. (Superceding DOD-STD-1838, Oct. 9, 1986.)

12. Boudier, Gerard; Gallo, Ferdinando; Minot, Regis; and Thomas, Ian: An Overview of PCTE and PCTE+. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Peter Henderson, ed., ACM Press, 1988, pp. 248–257.

13. Schulz, Steven E.: Frameworks: Debunking the Myths. *Electron. Design*, vol. 39, no. 6, Aug. 22, 1991, pp. 71–80.

14. Walker, Carrie K.; and Turkovich, John J.: Computer-Aided Software Engineering—An Approach to Real-Time Software Development. *A Collection of Technical Papers—Part 1, AIAA 7th Computers in Aerospace Conference*, Oct. 1989, pp. 10–19. (Available as AIAA-89-2961.)

15. *InQuisiX^{TM}—Software Reuse Library System.* Software Productivity Solutions, Inc., c.1991.

16. *ADAS—An Architecture Design and Assessment System for Electronic Systems Synthesis and Analysis—User's Manual, Version 2.5.* Cadre Technologies Inc., c.1988.

17. Young, Steven D.; and Wills, Robert W.: *Performance Analysis of a Large-Grain Data Flow Scheduling Paradigm.* NASA TP-3323, 1993.

18. Bernstein, Philip A.: Database System Support for Software Engineering—An Extended Abstract. *Proceedings—9th International Conference on Software Engineering*, IEEE Catalog No. 87CH2432-3, IEEE Computer Soc. Press, 1987, pp. 166–178.

19. Penedo, Maria H.; Ploedereder, Erhard; and Thomas, Ian: Object Management Issues for Software Engineering Environments—Workshop Report. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Peter Henderson, ed., ACM Press, 1988, pp. 226–234.

20. Lamb, David Alex: IDL: Sharing Intermediate Representations. *ACM Trans. Program. Lang. & Syst.*, vol. 9, no. 3, July 1987, pp. 297–318.

21. Atkinson, Malcolm P.; and Buneman, O. Peter: Types and Persistence in Database Programming Languages. *ACM Comput. Surv.*, vol. 19, no. 2, June 1987, pp. 105–190.

22. Sutton, Stanley M., Jr.: Appla/A: A Prototype Language for Software-Process Programming. Ph.D. Diss., Univ. of Colorado, July 1990.

23. Schmidt, Joachim W.: Some High Level Language Constructs for Data of Type Relation. *ACM Trans. Database Syst.*, vol. 2, no. 3, Sept. 1977, pp. 247–261.

24. Cockshott, W. Paul: *PS-ALGOL Implementations: Applications in Persistent Object Oriented Programming.* Ellis Horwood Ltd., 1990.

25. Zeil, Steven J.: Adding Persistence and Garbage Collection Within ADA. *Proceedings of the Ninth Annual National Conference on ADA Technology*, ANCOST, Inc., 1991, pp. 80–86.

26. Jipping, Michael J.; and Ford, Ray: Predicting Performance of Concurrency Control Designs. *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ACM Press, 1987, pp. 132–142.

27. Osterweil, Leon: Software Processes Are Software Too. *Proceedings—9th International Conference on Software Engineering*, IEEE Catalog No. 87CH2432-3, IEEE Computer Soc. Press, 1987, pp. 2–13.

28. Wild, Chris; and Maly, Kurt: Software Life Cycle Support—Decision Based Software Development. *Algorithms, Software, Architecture—Information Processing 92*, Volume I, J. Van Leeuwen, ed., Elsevier Science Publ., 1992, pp. 72–78.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1993	Technical Memorandum

4. TITLE AND SUBTITLE
Open Environments To Support Systems Engineering Tool Integration: A Study Using the Portable Common Tool Environment (PCTE)

5. FUNDING NUMBERS
WU 505-64-50-05

6. AUTHOR(S)
Dave E. Eckhardt, Jr., Michael J. Jipping, Chris J. Wild, Steven J. Zeil, and Cathy C. Roberts

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
L-17202

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA TM-4489

11. SUPPLEMENTARY NOTES
Eckhardt: Langley Research Center, Hampton, VA; Jipping: Hope College, Holland, MI; Wild and Zeil: Old Dominion University, Norfolk, VA.; Roberts: ICASE, Langley Research Center, Hampton, VA.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A study of computer engineering tool integration using the Portable Common Tool Environment (PCTE) Public Interface Standard is presented. Over a 10-week time frame, three existing software products were encapsulated to work in the Emeraude environment, an implementation of the PCTE version 1.5 standard. The software products used were a computer-aided software engineering (CASE) design tool, a software reuse tool, and a computer architecture design and analysis tool. The tool set was then demonstrated to work in a coordinated design process in the Emeraude environment. This paper describes the project and the features of PCTE used, summarizes experience with the use of Emeraude environment over the project time frame, and addresses several related areas for future research.

14. SUBJECT TERMS
Open environments; Software environments; Portable Common Tool Environment (PCTE)

15. NUMBER OF PAGES
15

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		